ProB User Manual

For ProB version 1.1.4

Declarative Systems and Software Engineering
University of Southampton, U.K.

and

Softwaretechnik und Programmiersprachen
University of Düsseldorf, Germany

©2005 M. A. Leuschel, M. Butler and S. Lo Presti

1 Typographic Conventions

Throughout this document a shorthand notation is used to refer to menus and options present in ProB. This convention, plus others, are detailed below.

1.1 Hints and Pitfalls

This text would indicate a potential problem or situation where ProB behaviour may be contrary to what is expected. Take careful note of the text of pitfalls.

This text indicates a helpful suggestion to improve ProB behaviour or user interaction with ProB.

1.2 Menus

The text from the title of a menu, a command in a menu, a preference, a part of the main ProB window, or certain operation names is displayed in a specific font. For example File, or State Properties. The name of an operation is displayed in italic font, for example *initialise_constants*. Text in the B notation is displayed with another font, for example `MACHINE`.

Navigation between the menus is shown using the symbol →. For example File→Open for the Open command of the File menu.

2 Introduction

ProB is a graphical animator and model checker for the B method [12] and [2]. The ProB homepage is at `http://www.ecs.soton.ac.uk/~mal/systems/prob.html`, where precompiled binaries, installation instructions and documentation are available.

2.1 Features

ProB enables a user to animate a B specification, either interactivally or automatically. It can also be used to systematically model check a B specification for errors. ProB is a tool useful in helping the user correct and understand its B specifications.

ProB covers a large part of the B notation, with features such as non-deterministic operations, `ANY` statements, operations with complex arguments, sets, sequences, relations, functions, lambda abstractions, set comprehensions, constants and properties, and many more. It has limited support for multiple machines and refinements. A summary of the B notation is available with the command About→Summary of the B syntax.

2.2 Current Limitations

ProB requires all variables to be typed explicitly, as it only has a very limited type inference. This is good specification style and is discussed in detail in chapter 4.

Other general limitations are:

`ASSERT` The use of the `ASSERT` statement within operations is not yet supported, but the use of the `ASSERTIONS` clause is supported. In the meantime, one can use the `PRE` construct instead;

`REC`, `LET`, `WHILE` These construct are not yet supported;

`closure` The transitive and reflexive closure operator `closure` is not supported.
However, the transitive closure operator `closure1` is supported, and hence one can translate an expression `closure(e)`, where e is a binary relation over the domain d, into the expression `closure1(e) \/ id(d)` . This limitation is due to the fact that one needs to infer the type of `e` to figure out the effect of `closure(e)` . The `closure` operator is thus not very transparent and potentially dangerous;

`STRING`, etc. Constructs that are specific to a B tool and are not standards (e.g., `STRING` or records from Atelier B) are not supported;

Definitions Definitions (from the `DEFINITIONS` clause) with arguments are not yet supported (due to a restriction in the parser.)

Note, one can use definitions without arguments. However, the *jbtools* parser can take a very long time to produce error messages if those definitions contain syntax errors.

2.2.1 Multiple Machines and Refinements

It is possible to use multiple machines with ProB. However, ProB does not yet support machine renaming and does not enforce the visibility rules. As far as the visibility rules are concerned, it is thus a good idea to check the machines in another B tool, such as Atelier B or the B-Toolkit.

2.3 Related Publications

A brief, 5-page description of ProB can be found on the ProB website [8]. A introductory paper describing the ProB system is available in [9]. You can also learn more about some of the state space visualisation features of ProB in [10], about the use of CSP in ProB in [3], and about the ProTest testing feature in [11].

2.4 Credits

ProB is maintained by Michael Leuschel. It is based on research and implementation effort by Michael Leuschel, Michael Butler, Carla Ferreira, Leonid Mikhailov, Edward Turner, Phil Turner, and Laksono Adhianto. Part of the research and development was conducted within the EPSRC funded projects ABCD [1] and iMoc [6].

ProB was developed using *SICStus Prolog* [13], but does not necessitate a SICStus Prolog license. It uses co-routining and finite domain constraint solving to make animation of B machines possible.

The input language of ProB is the XML encoding of the B notation. The translation between the two formats is done via the B-to-XML parser of Bruno Tatibouet's *jbtools* [14].

ProB also uses *Tcl/Tk* for the Graphical User Interface (a *Java* version is also available) and *dot*/*dotty* from the Graphviz package.

3 General Presentation

3.1 Graphical Interface

The ProB main window is displayed in Figure 3.1.

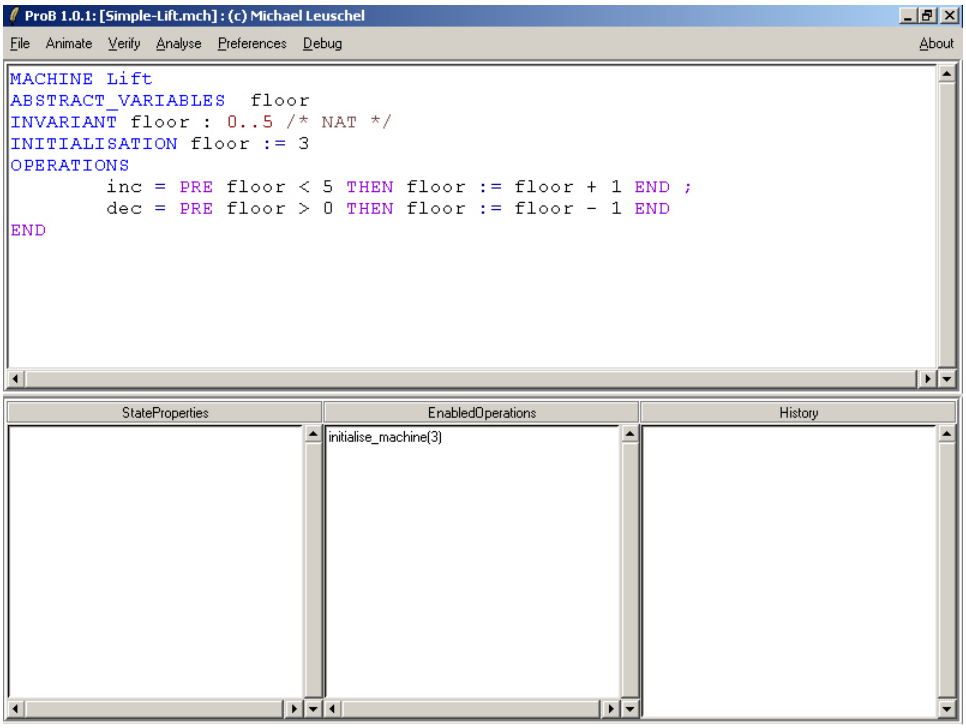


Figure 3.1: Main window of ProB

The menu bar contains the various commands to access the features of ProB. It includes the traditional File menu, with a submenu Recent Files to quickly access the files previously opened in ProB. Notice the two couples of commands Open\Save and Reopen\Save and Reopen, the latter reopening the currently opened file and reinitialising completely the state of the animation and the model checking processes. The About menu provides help on the tool, including a command to check if an update is available on the ProB website.

By default, ProB starts with a limited set of commands in the Beginner mode. The Normal mode gives access to more features and can be set in the menu Preferences→User Mode, see Figure 3.2.

Under the menu bar, the main window contains four panes:

- In the top pane, the specification of the B machine is displayed with syntax highlight, and can also be edited by typing directly in this pane;
- At the bottom, the animation window is composed of three panes which display, at the current point during the animation:
  - The current state of the B machine (State Properties), listing the current values of the machine variables;

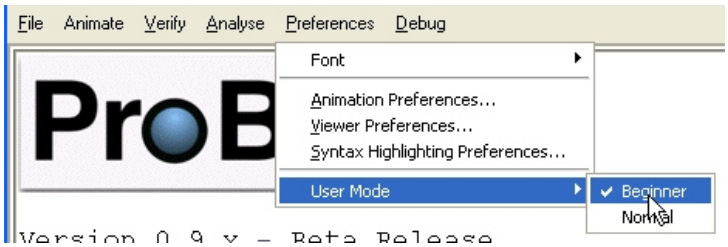


Figure 3.2: Setting the user mode

– The enabled operations (Enabled Operations), listing the operations whose preconditions and guards are true in this state;

– The history of operations leading to this state (History).

3.2 ProB Preferences

The Preferences menu enables to configure the various features of ProB. When ProB is started for the first time, it creates a file `prob_preferences.pl` that stores those preferences.

The submenu Font changes the font size of the B specification displayed in the main window.

The next three commands correspond to groups of preferences displayed in separate pop-up windows.

The command Animation Preferences... configures important aspects of ProB, relative to the animation and model checking of the B specifications. These preferences influence directly the way ProB interprets the B specification and are described in Sections 5.1.3, amongst others.

The command Graphical Viewer Preferences... enables to set the options of the visualisation tool used by ProB, and the shapes and colors used to display the nodes of the state space. This is explained in Section 5.2.1.

The command Syntax Highlight Preferences... enables to activate the syntax highlight of the B specification in the main window, and also to select the various colors corresponding to the syntactic elements of the B notation.

Changing in the animation preferences takes effect only for the next B machine opened in ProB (or necessitates that the currently loaded machine be reloaded).

4 Typing in ProB

ProB requires all constants and variables to be typed explicitly. The typing is essential for ProB's animation and verification algorithm and it is thus important to understand this aspect of ProB.

The undecidability of animating B machines is overcome by restricting animation to finite sets and integer ranges (see Section 4.4), while efficiency is achieved by delaying the enumeration of variables as long as possible (see Section 4.5).

4.1 What is a basic type in B

- `BOOL`
- `INT`
- Any name of a set introduced in a `SETS` clause or introduced as a parameter of the machine
- `POW` (τ) (power set) for τ bieng a type
- τ_1 `*` τ_2 (Cartesian product) for τ_1 and τ_2 being two types

4.2 What needs to be typed

Generally speaking, any constant or variable must be given a type for ProB to function properly. More precisely:

- Constants declared in the `CONSTANTS` clause must be typed in the `PROPERTIES` clause;
- Variables declared in the `VARIABLES` clause must be typed in the `INVARIANT`;
- Arguments of an operation must be typed in the precondition `PRE` or a top-level `SELECT` statement of the operation;
- Variables in universal or existential quantifications;
- Variables of set comprehensions must be typed in a conjunct of the body of the set comprehension. For example, {`xx | xx:NAT & xx>0 & xx<5`} is fine, but {`xx | xx>0 & xx<5`} is not;
- Variables of lambda abstractions must be typed in the predicate part of the abstraction. For example, `%yy.(yy:NAT|yy-1)` properly types the variable `yy`;
- Variables introduced in `ANY` statements must be typed in the `WHERE` part of the statement;
- For the moment the typing has to be repeated for refined operations as the typing is not extracted from ancestor specifications).

ProB will warn you if a variable has not been given a type. ProB may still function if some variables are not typed, but you run the risk of non-termination or floundering (question marks in the Enabled Operations pane), meaning that ProB was not able to resolve all constraints and cannot guarantee that the present state satisfies all constraints imposed by the B machine.

4.3 What does not need to be typed

The following declarations do not need to be typed, as ProB infers their type:

- `NAT`, `NATURAL`, `NAT1`, ... You can also use ranges like `1..10`;
- A constant or global variable that has already been typed;
- A constant or variable that is included (`<:`) in a typed set;
- A relation over A,B is equivalent to `POW (A*B)` , and similarly for the functions `-+->`, `->`, `+->>`, `-->>`, `>+>`, `>->`, `>+>>`, `>->>`;
- A sequence over `A`, which is equivalent to `POW (INT*A)` .

The type inference of ProB is more limited than the full type inference algorithm of B tools like Atelier B. It is good specification style to explicitly type all your variables, rather than relying too much on a sophisticated type inference (as this may hide problems in the specification).

The Analyse→Show Typing command reveals the typing that ProB has inferred for your constants and global variables.

4.4 Restriction on the Domains of the Variables

Animating and verifying a B specification is in principle undecidable. ProB overcomes this by requiring that the domain of the variables is finite (i.e., with finitely many values) or integer. This ensures that the state space has finite size. Typing of the B specification ensures this restriction.

In the B specification, a set is either defined explicitely, thus being a finite domain, or its definition is deferred. In the later case, the user can indicate the size of the set `mySET` (without defining its elements) by creating a macro in the `DEFINITIONS` clause with the name `scope_mySET` and a value specified as a range, e.g. `1..12`. The macros with the prefix `scope_` will be used by ProB and do not modify the B specification. If the size of the set is unspecified, ProB considers the set to have a default size. The value for the default size is defined in the Preferences→Animation Preferences... preference window by the preference Size of unspecified sets in `SETS` section.

The B method enables to specify the size of a set with the `card` operator but this form of constraint is not yet supported by ProB.

As a consequence of the restriction on the domains of the variables, ProB makes no distinction between the implementable integers `INT` and `INTEGER`, nor between `NAT` and `NATURAL`, nor between `FIN` and `POW`.

4.5 Enumeration in ProB

The typing information is used by ProB to enumerate the possible values of a constant or a variable whenever a specification does not narrow down that value to a single value.

For example, if you write `xx:NAT & xx=1` ProB does not have to resort to enumeration as the `xx=1` constraint imposes a single possible value for `xx`. However, if you write `xx:NAT & xx<3` ProB will enumerate the possible values of `xx` in order to find those that satisfy the constraints imposed by the machine (here `0,1,2`).

ProB will use the constraints to try to cut down the enumeration space, and will resort to enumeration usually only as a last resort. So something like `xx:NAT & xx<10 & x>2 & x=5` will not result in enumeration.

The enumeration range for integers is controlled by two preferences in the Preferences→Animation Preferences... preference window: MinInt, used for expressions such as xx::INT, and MaxInt, used for expressions such as xx::NAT preferences. Nevertheless, writing `xx: NAT & xx =55` puts the value 55 in `x` no matter what MaxInt is set to, as no enumeration is required.

5 Animation and Visualisation

We present in this chapter the features of ProB for animating B machines and visualising the state space of the machine.

5.1 Animation

The animation facilities of ProB allow users to gain confidence in their specifications. These features are user-friendly as the user does not have to guess the right values for the operation arguments or choice variables, and he uses the mouse to operate the animation.

5.1.1 Basic Animation

When the B specification is opened, the syntax checker analyses it and, if a syntax error is detected, it is then reported, highlighted in yellow in the specification (the user can choose to display a verbose error message obtained from the *jbtools* parser). Furthermore, if the B specification contains features of B that are not supported by ProB or constraints that are not satisfiable, an appropriate message is displayed. When these checks are passed, the B machine is loaded, but it has no state yet.

ProB will then display the operations that can be performed in the Enabled Operations pane. These operations can be of two types described below.

Operations from the B Machine

These operations are the ones whose preconditions and guards are satisfiable in the current state. The parameter values that make true the precondition and guard are automatically computed by ProB, and one entry for the operation is displayed in the Enabled Operations pane for each group of parameter values. Each parameter value is displayed as a set between curly brackets, and the group of parameter values are enclosed between brackets after the operation name.

The computation of the parameter values greatly facilitates the work of the user, as he does not have to enumerate the possible values and check the preconditions and guards. This computation process involves trying to solve the various constraints imposed on the parameter values in the preconditions and guards.

If no operation is enabled, the state of the B machine corresponds to a deadlock

Virtual Operations

There are three particular operations that correspond to specific tasks performed by ProB:

- *initialise_constants*
  This virtual operation corresponds to the assignment of values to the constants of the B machine. These values must satisfy the `PROPERTIES` clause. ProB automatically computes the possible values and displays one *initialise_constants* virtual operation for each possible group of of constant values. If the `PROPERTIES` clause is not satisfiable, an error message is displayed.

- *initialise_machine*
  This virtual operation plays the same role as *initialise_constants* but for initial values of the variables (clauses `VARIABLES` and `INITIALISATION`) instead of constants. If the `INITIALISATION` clause does not satisfy the constraints in the `INVARIANT` clause, an error message is displayed.
- BACKTRACK
  This virtual operation is used during animation to go back to the previous state, that is the one before the operation that was selected to reach the current state. This operation enables the user to explore interactively the behaviour of the B machine.

Animating the B machine

If the B machine has constants, one or several *initialise_constants* operations are displayed. The user selects one of these operations, then the corresponding values of the constants are displayed in the State Properties pane and the selected *initialise_constants* operation is displayed in the History pane. In the State Properties pane, functions and relations are displayed by indicating each of their tuples on a different line.

At that point during the animation (also reached directly if the B machine has no constants), ProB displays one or several *initialise_machine* operations. The user selects one of these operations, and then the machine is in its initial state. The initial values of the variables are displayed in the State Properties pane and the *initialise_machine* operation selected is displayed in the History. From that moment on, an indicator of the status of the invariant is displayed at the top of the State Properties pane and the BACKTRACK operation is displayed at the bottom of the Enabled Operations pane. The invariant status indicator is `invariant_ok` if the invariant is satisfied or `invariant_violated` if the invariant is violated.

From there, the user selects operations among the enabled ones. If the selected operation is BACKTRACK, the last selected operation is removed from the top of the History pane and the previous state is displayed in the State Properties pane. If the operation was not BACKTRACK, it is added to the History pane, the effect of the operation are computed and the state is updated in the State Properties pane.

5.1.2 The Analyse menu

At each point during the animation process, several useful commands displaying various information on the B machine are available in the Analyse menu.

The Compute Coverage command opens a window that displays three groups of information:

`NODES` This is the number of nodes (i.e. states) explored so far; there are three kinds of nodes:

- `live`: states already computed by ProB;
- `deadlocked`: states where the B machine is deadlocked;
- `invariant_violated`: states where the invariant is violated;
- `open`: states that are reachable from the live nodes by an enabled operation.

`COVERED_OPERATIONS` This is the number of operations that have been enabled so far, including the *initialise_machine* operations.

`UNCOVERED_OPERATIONS` This is the names of the operations that have not been enabled so far.

The Analyse Invariant command opens a window displaying the truth values of the various conjuncts of the invariant of the B machine, while the Analyse Properties command plays the same role but for the constant properties and the Analyse Assertions plays this role for the assertions.

5.1.3 Preferences of the Animation

The animation process in ProB can be configured via several preferences set in the preference window Preferences→Animation Preferences....

First, the preference Show effect of initialisation and setup constants in operation name toggles the display of the values of the constants and the initial values of the variables when the corresponding virtual operations are shown in the Enabled Operations pane.

The preference Nr of Initialisations shown determines the number of maximum *initialise_machine* operations that ProB should compute. Similarly, the preference Max Number of Enablings shown sets the maximum number of groups of parameter values computed for each operation of the B machine.

The preference Check invariant will toggle the display of the invariant status indicator in the State Properties pane.

5.1.4 Other Animation Features

Several other commands are provided by ProB in the Animate menu for animating B specifications.

The Reset command will set the state of the machine to the root, as if the machine has just been opened, i.e. when the *initialise_constants* or *initialise_machine* operations are displayed in the Enabled Operations pane.

The Random Animation(10) command operates a sequence of 10 randomly chosen operations from the B specification. The variant Random Animation enables to specify the number of operation to operate randomly.

In the File menu, the command Save State stores the current state of the B machine, which can then be reloaded with the command Load Saved State.

5.2 Visualisation

ProB provides several user-friendly visualisation features to help the user to analyse and understand the behaviour of his B specification. This feedback is very beneficial to the understanding of the B specification since human perception is good at identifying structural similarities and symmetries. For more information on this particular topic, the reader can refer to this paper [10].

The visualisation features are in the Animate menu, and comprise the command View Visited States and all the commands of the submenu View. It is important to understand that those commands operate on the state space computed by ProB at the current point during the animation. Each time the user animates the B specification, the state space computed by ProB can be expanded if the selected operations lead to states not already computed by ProB.

The command View Visited States displays in a separate window a diagram corresponding to the state space currently explored by the animation, as shown in Figure 5.1.

Graph Nodes

ProB displays the state space as a graph whose nodes correspond to states that are differentiated by their shapes and colors, and arcs correspond to operations. The operations are all those that are displayed in the Enabled Operations pane except BACKTRACK, which is only useful for animation. Four types of nodes are visualised in ProB:

root Point before the B machine is initialised, when it has no state;

current The current state during the animation;

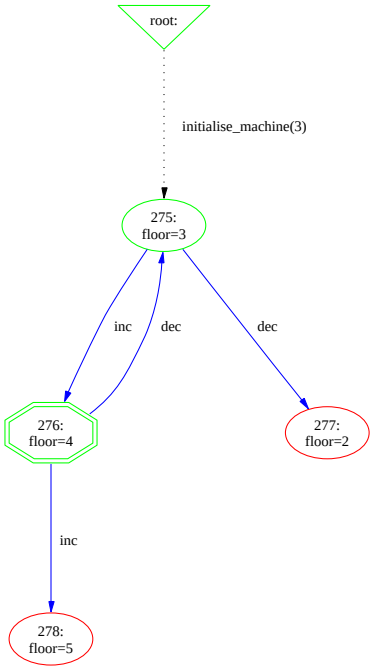


Figure 5.1: Visualising the State Space

normal The states that have been already computed during the animation;

open The states that are reachable from the normal states by an enabled operation.

In addition to its type, a node can indicate that it corresponds to an invariant violation, displayed by a color filling as is shown on Figure 5.2.

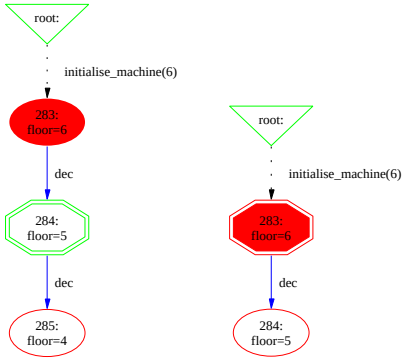


Figure 5.2: Normal and current nodes with invariant violation

Other Visualisation Commands

The menu View Visited States→View contains several other commands that provide useful views on the state space. The command Shortest Trace to Current State displays the shortest sequence of nodes in the state space starting from the root node and leading to the current node. The command Current State displays the current node and its successor nodes.

The three next commands in the menu View Visited States→View provide a means to display a simplified version of the state space. A more detailed explanation is given in this paper [10].

The command Reduced Visited States displays a state space where nodes sharing the same output arcs are collapsed into one node. The command Reducted Deterministic Automaton of Visited States removes the non-determinacy of the state space graph. The command Select Operations & Arguments for Reduction is used to specify which operations and arguments are affected by the previous transformations.

The two last commands of the View submenu display subgraphs of the state space. The command Subgraph for Invariant Violation displays the subgraph of nodes which violate the invariant, while the command Subgraph of nodes satisfying GOAL displays the subgraph where goals (discussed in 6.3) are satisfied.

5.2.1 Preferences of the Visualisation

Many aspects of the visualisation can be configured in the Graphical Viewer Preferences... preference window of the Preferences menu. This includes changing the shapes and colors of the various nodes (using the notation of the *dot* tool, see [4] and [5]). For selecting the colors, a color picker is available via the button Pick. The user can also select which labels to display on the nodes (value of variables) and arcs (operation arguments and return value of functions), and their font size.

The default graph viewer in ProB is dotty, from the Graphviz package. ProB enables the user to display the graph using a PostScript viewer by setting to true the preference Use Postscript Viewer in the Graphical Viewer Preferences.... The PostScript file is generated by the *dot* tool. The path to the PostScript viewer can be set in the Path/Command for Postscript Viewer preference. The Pick button can be used to select the path.

All paths to files and folders should use the / character to separate the folders and should be absolute.

Using a postscript viewer rather than dotty has several advantages and several drawbacks. Firstly, the assignment of node shapes and colours is more accurately realised by dot (and therefore PostScript). Dotty on the other hand is much easier to use when state spaces are large, notably thanks to the *birdseye view*. A PostScript viewer also has the advantage that the PostScript file may be used to capture visualisations for publication purposes.

At present, the distinction between using a PostScript viewer as opposed to dotty comes down to the difference in functionality between the GraphViz utilities dot and dotty. The main differences with respect to visualisation in ProB are are:

- For Postscript: Support for more visualisation shapes (for example, the shape *doubleoctagon* appears as a box on dotty);
- Against PostScript: *dot* does not support the addition of shapes to arcs. With moderately large graphs, Dot may put nodes outside of the printable or viewable area. Examining large graphs in a PostScript viewer may be slow (it may be awkward using pan and zoom). Support for less information on arcs (for example, dotted lines).
- For Dotty: Graphs can be modified. Dotty includes a *birdseye* viewer which is useful for examining large graphs.
- Against Dotty: Dotty may crash if the graph is too big or complex (and on Solaris and Linux, if non-standard mouse buttons are used).

6 Temporal Model Checking

6.1 Introduction

By manually animating a B machine it is possible to discover problems with the specification, such as invariant violations or deadlocks. The ProB model checker can do this exploration systematically and automatically. It will alert you as soon as a problem has been found, and will then present the shortest trace (within currently explored states) that leads from an initial state to the error.

The model checker will also detect when all states have been explored, and can thus also be used to formally guarantee the absence of errors. This will obviously only happen if the state space is finite, but the model checker can also be applied to B machines with infinite state spaces and will then explore the state space until it finds an error or runs out of memory.

6.2 Basics

During the initial draft of a specification, it is often useful to utilise the model checker to determine if there are any invariant violations or deadlocks. A model checker automatically explores the (finite or infinite)state space of a B machines.

Recall, that the `INVARIANT` specifies both the types of variables and also may include relationships that must hold over them in all situations, that is properties of a B machine that are immutable and must be permanently established.

Since a graph of the visited states and the transformations (operations) between them is retained, it is therefore possible to produce traces (sequence of operations) of invariant violations when they are detected. Such a trace is called a "counter-example" and is useful in determining where and in what circumstances a specification contains errors.

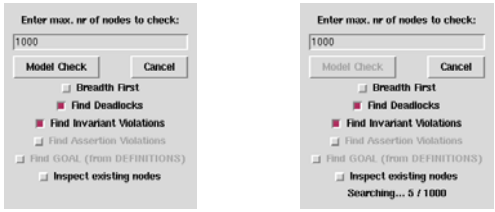
Notice that if the model checker does not find any invariant violations or deadlocks when a traversal of the exhaustive state space has been performed, this does not imply that the specification is a correct specification. This should be understood as the fact that, given the initialisation stated and the model checker preferences set at the time of the check, no errors were found. The main difference is that the ProB animation preferences (such as number of initialisations and enablings, and size of unspecified sets) may be set too low for the exhaustive state space to be covered by the model checking.

As a final caveat, it is not possible to check a machine with an infinite state space. Anecdotal evidence does suggest however, that the model checker does find errors quite quickly. On that basis, it remains a useful tool even for specifications whose state space is infinite.

6.2.1 Using the Temporal Model Checker (TMC)

To model check a B machine loaded in ProB, launch the command from Verify→Temporal Model Check... (see Figure 6.1a). The entry Enter max nr. of nodes to check in the pop-up window indicates the number of states that ProB should compute. This computation is done by exploring the state space in a depth first manner, unless the setting Breadth First is toggled on. The two toggles Find Deadlocks and Find Invariant Violations add to this computation a verification of the corresponding property each time a new node is computed.

The temporal model checker is started by clicking on the Model Check button. When the TMC computes and searches the state space, a line prefixed with Searching... at the bottom of the pop-up window will update in realtime the number of nodes that have been computed,



(a) TMC Settings

(b) TMC Progress through state space

Figure 6.1: Temporal Model Checker settings and progress

until the number of nodes indicated in the Enter max nr. of nodes to check entry is reached or a node violating one of the properties verified has been found. The user can interrupt the model checking at any time by pressing the Cancel button.

Incremental Model Checking

It is important to understand that the computation of and search in the state space is an incremental process. The model checking can be done by running the TMC several times; if the number of nodes specified in the entry max nr. of nodes to check is less that the size of the state space that remains to be checked. If the model checking is done in several steps, the end of the model checking step is indicated by the line No error so far, ... nodes visited at the bottom of the pop-up window, unless a violation of the properties (deadlockfreeness, invariant) are found. Between each model checking step, the user can execute the various commands of ProB, notably those of the Analyse menu to display information on the state space (see Section 5.1.2) and the visualisation features (see Section 5.2).

By default, each model checking step starts from the open nodes of the last computed state space. This means that a change in the settings of the TMC between two steps does not affect the non-open nodes (thos already computed), unless there is an alternative path to an already computed node. This behaviour can be changed by toggling on the Inspect Existing Nodes setting. This forces ProB to reevaluate the properties set to be verified on the state space previously computed. Keep in mind that unless the Inspect Existing Nodes setting is on, the change of the TMC settings may not affect the state space already computed.

Results of the Model Checking

When the state space has been computed by ProB, the pop-up window is replaced by a dialog window stating No error state found. All new nodes visited. If ProB finds a node where one of the property that was checked is not verified, it displays a similar pop-up window but with the text Error state was found for new node: invariant_violation or Error state was found for new node: deadlock. Then, ProB displays in the animation panes the state of the B machine corresponding to the property violation. From there, the user can examine the state in the State Properties pane, the enabled operations (including the BACKTRACK virtual operation to go back to the valid state that lead to the property violation) in the Enabled Operations pane, and the trace (sequence of operations) leading to the invalid state in the History pane.

Preferences of the Model Checking

The preferences Nr of Initialisations shown and Max Number of Enablings shown (described in Section 5.1.3) affect the model checking in exactly the same way as they do for the animation. These preferences are particularly important for the model checking, as setting the number of initialisations or the number of enablings too low will lead to computing and searching a state space that is too small. On the other hand, the user may have to set these preferences to a lower value if the exhaustive state space is too big to be covered by ProB.

Once the state space of the B specification has been computed by ProB, the commands from the Analyse menu (see Section 5.1.2) and the visualisation features (see 5.2) are then very useful to examine the state space. The features used to visualise a reduced state space are particularly useful as examining a huge state space may not yield to interesting observations due to excessive cluttering [10].

6.2.2 Machines with Invariant Violations and Deadlocks

When the two properties (invariant and deadlockfreeness) are checked and a state violates both , only the invariant violation is reported. In that situation, the deadlock can be observed either from the Enabled Operations (as only the BACKTRACK virtual operation is enabled), or from the state space graph (as the current node has no successors).

During the model checking of a specification which contains both of these errors, it is often useful to be able to focus exclusively upon the detection of one type of error. For example, since an invariant violation is only reported the first time it is encountered, subsequent invocations of the TMC may yield deadlocks whose cause is the invariant violation. This is done by toggling off the corresponding settings of the temporal model checker pop-up window.

Turning off invariant violation and deadlock detection will simply compute the entire state space until the user press the Cancel button or the specified number of nodes is reached.

Two Phase Verification

If the state space of the specification can be exhaustively searched, check for deadlocks and invariant violations in two phases. To do this, set Inspect Existing Nodes off and set one of Check for Deadlocks or Check for Invariant Violations on and the other off. Perform the temporal model check until either all the deadlocks (resp. invariant violations) are detected or shown to be absent. To recheck the whole state space either turn on the option Inspect Existing Nodes on, or else purge the states space already computed by reopening the machine. Now deselect the previous property to be checked, and select the alternative property for checking. Now perform a temporal model check again searching for a violation of the second property.

At any time during animation and model checking, the user can reopen the the B specification to purge the state space.

Interleaved deadlock and invariant violation checking

If you wish to determine if an *already encountered* invariant violation is also a deadlocked node, turn the option Inspect Existing Nodes on, turn Detect Invariant Violations off, and ensure that Detect Deadlocks is on. Performing a temporal model check now will traverse the state space including the previously found node that violates the invariant.

Enabling Inspect Existing Nodes will continually report the first error it encounters until that error is corrected.

6.3 Specifying Goals and Assertions

The `ASSERTIONS` clause of a B machine enables the user to define predicates that are supposed to be deducible from the `INVARIANT` or `PROPERTIES` clauses. If the B specification opened in ProB contains an `ASSERTIONS` clause, the TMC pop-up window enables to check if the assertion can be violated. If enabled and a state corresponding to the violation of the assertion is found, a message Error state was found for new node: assertion_violation is displayed, and then ProB displays this state in the animation panes.

A feature that is similar to the assertions is the notion of a *goal*. A goal is a macro in the `DEFINITIONS` section whose name is `GOAL` and whose content is a predicate. If the B specification defines such a macro, the TMC pop-up window enables to check if a state satisfies this goal. If enabled and a state corresponding to the goal is found, a message State satisfying GOAL predicate was found is displayed, and then ProB displays this state in the animation panes.

It is also possible to find a state of the B machine that satisfies such a goal without defining it explicitely in the B specification. The Verify→Advanced Find... command enables the user to type a predicate directly in a text field. ProB then searches for a state of the state space currently computed that satisfies this goal.

7 Constraint Based Checking

7.1 Introduction

ProB also offers an alternative checking method, inspired by the Alloy [7] analyser. In this mode of operation, ProB does not explore the reachable states starting from the initial state(s), but checks whether applying a *single* operation can result in a property violation (invariant and assertion), independently of the particular initialisation of the B machine. This is done by symbolic constraint solving, and we call this approach *constraint based checking* (another sensible name would be *model finding*).

7.2 Difference between Constraint Based Checking and Model Checking

Model checking tries to find a sequence of operations that, starting from an initial state, leads to a state which violates a property. *Constraint based checking* tries to find a state of the machine that satisfies the property, but where we can apply a single operation to reach a state that violates this property.

If the constraint based checker finds a counter-example, this indicates that the B specification may contain a problem. The sequence of operations discovered by the constraint based checker leads from a valid state to a property violation, meaning that the B machine may have to be corrected. The valid state is not necessarily reachable from a valid initial state. This situation further indicates that it will be impossible to prove the machine using the B proof rules.

7.3 Commands of the Constraint Based Checker

These commands are only accessible in the Normal mode of ProB (see Section 3.1) and are in the submenu Verify→Constraint Based Checking.

The command Constraint Search for Invariant Violations will execute the constraint based checking described above, cheking for invariant violations. The process stops as soon as a counter-example has been found and then displays the counter-example in the animation panes. The command Constraint Search for Invariant Violations... enables the user to specify which particular operation leading from the valid state to the invariant violation of the B specification should be considered during the constraint based checking.

The Constraint Search for Assertion Violations command executes the constraint based checking looking for assertion violations, while the command Constraint Search for Abort Conditions executes it looking for abort conditions.

8 Advanced Features

The advanced features of ProB are available in the Normal mode of ProB (see Section 3.1).

8.1 Refinement Checking

The command Analyse→View Machine Hierarchy displays graphically the machine hierarchy. Each node in this graph corresponds to a machine and indicates its name, variables and operations. For specifications involving several machines, the graph displays their relationships by arcs labelled with the name of the clause corresponding to the relationship (e.g. `INCLUDES`).

To be completed.

8.2 Use of B and CSP

This features is described in this paper [3].

When using this feature, the name of a set should not begin with an upper-case letter, because it would be interpreted as variables in CSP.

To be completed.

8.3 ProTest

This features is described in this paper [11].

To be completed.

Appendix A ProB Licence

The ProB licence is shown in the main ProB windows whenever ProB starts up. Revisions to the licence will be shown whenever you update ProB.

```
Version 1.1.4
Released on July 2 2004

ProB: A B-animator and model checker
------------------------------------

(C) 2000-2004 Michael Leuschel, DSSE, University of Southampton
All rights reserved.
Free for non-commercial, academic use or evaluation purposes.
For commercial use, contact the author (http://www.ecs.soton.ac.uk/~mal).


ProB comes with ABSOLUTELY NO WARRANTY OF ANY KIND !
This software is distributed in the hope that it will be useful
but WITHOUT ANY WARRANTY. The author(s) do not accept responsibility
to anyone for the consequences of using it or for whether it serves
any particular purpose or works at all. No warranty is made about
the software or its performance.


For updates and news check:
  http://www.ecs.soton.ac.uk/~mal/systems/prob.html
  http://www.ecs.soton.ac.uk/~mal/systems/ProB_Download/Updates/

ProB uses state-of-the-art Prolog technology (co-routining, finite
domain constraint solvers, PILLOW XML package,...) to achieve symbolic debugging,
constraint-based and temporal-logic based model checking.
The tool is partly being developed within the EPSRC grants iMoc and ABCD.

Development, Copyright and Intellectual Property Rights:
B-Kernel & Model Checker:
Michael Leuschel
TCL/TK Interface:
Michael Leuschel, Laksono Adhianto
"XML to Prolog" Parser, Compiler & Further Extensions:
Michael Butler, Carla Ferreira,
Michael Leuschel, Stephane Lo-Presti, Leonid Mikhailov,
Ed Turner, Phil Turner

Open Source "B to XML" Java Parser + Type Checker:
    Bruno Tatibouet
    (see http://lifc.univ-fcomte.fr/~tatibouet/JBTOOLS/ for more details)


Known limitations:
==================

Many:
 - Unsupported features: complex definitions, some operators, WHILE loops...
    (see included machines for which features are actually supported)
```

```
 - not yet optimised for speed using partial evaluation
 - recursive set comprehension  ( ss = {xx |  ... xx: ss ...}) may not always work properly


To visualize state space,...: dot from AT&T's GraphViz package has to be installed


See the FAQ.txt file for troubleshooting and frequently asked questions.
```

Bibliography

[1] Automated validation of business critical systems with component based designs. http://www.ecs.soton.ac.uk/~phh/abcd/.

[2] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 1996.

[3] M. Butler and M. Leuschel. Combining CSP and B for specification and property verification. Technical report, School of Electronics and Computer Science, University of Southampton, 2005. http://eprints.ecs.soton.ac.uk/10388/.

[4] Node shapes. http://www.graphviz.org/cvs/doc/info/shapes.html.

[5] Color names. http://www.graphviz.org/cvs/doc/info/colors.html.

[6] imoc: Infinite state model checking using abstract interpretation and model checking. http://www.ecs.soton.ac.uk/~mal/ISM.html.

[7] D. Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[8] M. Leuschel and M. Butler. The proB animator and model checker for B - a tool description. http://www.ecs.soton.ac.uk/~mal/systems/prob_tooldescription.pdf.

[9] M. Leuschel and M. Butler. ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003. http://eprints.ecs.soton.ac.uk/archive/00008341/.

[10] M. Leuschel and E. Turner. Visualising larger state spaces in proB. In H. Treharne, S. King, M. Henson, and S. Schneider, editors, *ZB 2005: Formal Specification and Development in Z and B*, LNCS 3455. Springer-Verlag, 2005.

[11] M. Satpathy, M. Leuschel, and Butler. ProTest: An automatic test environment for B specifications. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 111:113–136, 2005.

[12] S. Schneider. *The B-Method: An Introduction*. Palgrave, 2001.

[13] Sicstus prolog. http://www.sics.se/isl/sicstuswww/site/.

[14] B. Tatibouet. jbtools. http://lifc.univ-fcomte.fr/~tatibouet/JBTOOLS/index_en.html.